TESTE DE SOFTWARE

AULA 5

Profª Maristela Regina Weinfurter Teixeira

CONVERSA INICIAL

Testes de Homologação

Todo desenvolvedor de software entende que falhas de software podem causar sérios danos a empresas, pessoas e até mesmo à sociedade como um todo. Embora os desenvolvedores de software tenham sido os principais responsáveis pela construção de sistemas de software, hoje eles também são responsáveis pela qualidade dos sistemas de software que produzem.

As comunidades de desenvolvedores produziram várias ferramentas de classe mundial para ajudar os desenvolvedores com seus testes, incluindo JUnit, AssertJ e Selenium. Aprendemos a usar o processo de escrita de testes para refletir sobre o que os programas precisam fazer e como podemos obter feedback sobre o design do código. Também aprendemos que escrever um código de teste é desafiador, de modo que é essencial prestar atenção à qualidade do código de teste para a evolução harmoniosa do conjunto de testes. Finalmente, sabemos quais são os bugs mais comuns e como procurá-los.

Mesmo com toda evolução na percepção sobre os testes e a qualidade de software, muitos profissionais argumentam que os testes são uma ferramenta de feedback, devendo ser usados principalmente para ajudar no próprio autodesenvolvimento como profissional. Embora isso seja verdade, os testes também podem ajudá-lo a encontrar bugs. Afinal, é para isso que serve o teste de software: encontrar bugs!

Um fato triste é que a maioria dos desenvolvedores não gosta de escrever testes. Os motivos para isso são inúmeros, por exemplo: escrever código direto em produção é mais divertido e desafiador; testar software consome muito tempo; somos pagos para escrever código de produção. Os desenvolvedores também superestimam o tempo que gastam em testes. Nosso objetivo aqui é deixar claro, como estudantes:

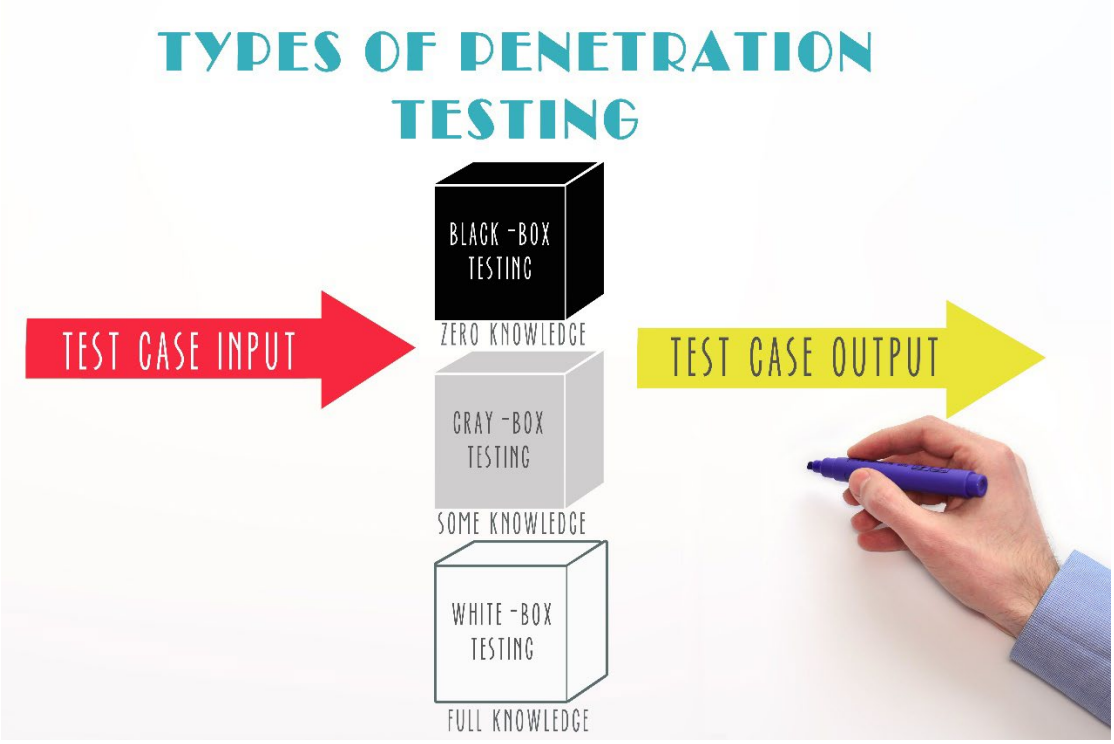
- como desenvolvedor, é sua responsabilidade garantir a qualidade do que você produz;
- os testes são as únicas ferramentas para ajudá-lo com essa responsabilidade;
- se você seguir uma coleção de técnicas, poderá testar o seu código de maneira eficaz e sistemática.

Eficaz e sistematicamente são palavras mais bem compreendidas no dia a dia, quando enfrentamos desafios de desenvolvimento e testes de software, em busca da garantia de qualidade. Para conseguir essa eficácia e eficiência, precisamos compreender todo o processo de homologação de testes, com todas as variedades, classificações e tipos de testes existentes, para que seja possível, além de aprender a usá-los, escolhê-los de forma otimizada para os nossos projetos.

TEMA 1 – TIPOS DE TESTES: CAIXA BRANCA (ESTRUTURAIS)

O teste de caixa branca baseado em fluxo de controle é a maneira mais barata de encontrar certos bugs de codificação, porque muitos deles são erros em fluxos de controle que existem inteiramente dentro de uma linha ou de algumas linhas de uma unidade de código. Os primeiros testes de unidade do tipo caixa branca feitos por programadores podem aumentar (e muito) a qualidade do sistema, reduzindo os custos de qualidade posteriormente no projeto (Black, 2007). A figura a seguir ilustra as diferenças entre os testes de tipo caixa branca e caixa preta.

Figura 1 – Caixa branca e caixa preta



Crédito: stel design/Shutterstock.

O teste de software de caixa branca é uma técnica de teste de software concentrada na estrutura **interna** do software que vai ser testado. O objetivo é fazer uma verificação do código e de suas estruturas internas, com a intenção de compreender se tudo está funcionando de acordo com as especificações e se a implementação é eficiente.

Esses testes ainda incluem verificações de unidade, de integração, validação das estruturas e outros testes para validação da qualidade da **funcionalidade do código**. A equipe de teste pode usar ferramentas de análise estática de código, bem como ferramentas de depuração, com a finalidade de identificar problemas de performance, segurança e correção de bugs.

Ao contrário dos testes de caixa preta, que se concentram nas funcionalidades visíveis do sistema, os testes de caixa branca permitem que a equipe de teste verifique o sistema em um nível mais **profundo**, o que pode levar a uma melhor compreensão da qualidade do código e à identificação preventiva de problemas.

O profissional de testes talvez não considere que essa seção é aplicável ao seu trabalho diário. No entanto, encorajo você a dar uma olhada rápida nos testes de caixa branca, considerando fluxo de dados. Uma familiaridade básica com boas práticas de testes de unidade pode ajudar a desenvolver uma conversa inteligente com os seus colegas programadores, avaliando a cobertura do teste de unidade.

Por outro lado, o desenvolvedor encontra uma fonte importantíssima nos conteúdos relacionados a testes do tipo caixa branca. Os testes de caixa branca se baseiam na forma como o sistema funciona internamente. Para construí-los, podemos examinar o código para encontrar os fluxos de controle em cada unidade. Também podemos examinar o código e as estruturas de dados para encontrar os fluxos de dados em cada unidade, ou ainda entre as unidades. Além disso, é importante examinar as interfaces do usuário (UI) do programa aplicativo, com as funções de membro de classe ou os métodos de classe para descobrir como um componente, uma biblioteca ou um objeto interage com o restante do sistema.

Testes baseados em código ou baseados em fluxo de controle geralmente são projetados para alcançar um determinado nível de cobertura de código. Existem sete maneiras principais de medir a cobertura de código:

- **Cobertura da declaração**: atingimos 100% de cobertura de instrução quando executamos todas as instruções.
- **Cobertura de ramo (ou decisão)**: alcançamos 100% de cobertura de filial quando são tomadas todas as ramificações (ou decisões) em cada sentido. Para instruções *if*, é necessário garantir que a expressão que controla a ramificação avalie verdadeiro e falso. Para instruções *switch/case*, precisamos cobrir cada caso especificado, com pelo menos um caso não especificado ou um caso padrão.
- **Cobertura de condição**: algumas decisões de ramificação em programas não são feitas com base em uma única condição, mas em várias condições. Atingimos 100 por cento de cobertura de condição quando avaliamos o comportamento com cada uma das condições verdadeiras e falsas. Por exemplo, se a expressão condicional que controla uma instrução *if* for (A>0) && (B<0), será necessário testar (A>0) verdadeiro e falso e (B<0) verdadeiro e falso. Isso pode ser feito em dois testes: verdadeiro && verdadeiro e falso && falso.
- **Cobertura multicondição**: quando as condições compostas são aplicadas à ramificação, podemos ir além da cobertura de condição para a cobertura de multicondição, exigindo o teste de todas as combinações possíveis de condições. Para continuar nosso exemplo com a expressão condicional que controla uma instrução *if*, temos (A>0) && (B<0), e assim precisamos testar todas as quatro combinações possíveis: verdadeiro && verdadeiro, verdadeiro && falso, falso && verdadeiro e falso && falso.
- **Cobertura de decisão multicondição**: para linguagens como C++, em que as condições subsequentes podem não ser avaliadas, dependendo das condições anteriores, 100% de cobertura de condição nem sempre faz sentido. Nesse caso, cobrimos as condições que podem afetar a decisão tomada em termos de fluxo de controle. Para finalizar o nosso exemplo, em que a expressão condicional que controla uma instrução *if* é (A>0) && (B<0), devemos testar três combinações: true && true, true && false e false && true. A combinação extra, false && false, agora é desnecessária porque a segunda condição não pode influenciar a decisão.
- **Cobertura de loop**: para caminhos de loop, devemos iterar cada loop zero, uma vez e várias vezes. Idealmente, repetimos o loop do número

máximo de vezes possível. Algumas vezes é possível prever o número máximo de vezes que um loop itera; às vezes não é possível.

- **Cobertura do caminho**: atingimos 100 por cento de cobertura de caminho quando tomamos todos os caminhos de controle possíveis. Para funções com loops, a cobertura do caminho é muito difícil.

Até certo ponto, podemos classificá-los em ordem crescente de níveis de cobertura. Se atingimos a cobertura de caminho, com certeza alcançamos cobertura de loop, ramificação e instrução (mas não vice-versa). Se alcançamos a cobertura da filial, com certeza atingimos a declaração de cobertura (mas não vice-versa). Se obtivemos a cobertura de multicondição, temos a garantia de ter alcançado a decisão multicondição e a cobertura condicional (mas não vice-versa). Observamos que a cobertura de caminho não garante a cobertura multicondição; a cobertura multicondição também não garante a cobertura de caminho.

Figura 2 – Programa que calcula fatoriais

```
main(){
  int i, n, f;

  printf("n = ");
  scanf("%d", &n);

  if (n < 0) {
    printf("Invalid: %d\n", n);
    n = -1;
  } else {
    f = 1;

    for (i = 1; i <= n; i++) {
        f*=i;
    }

    printf("%d! = %d.\n", n, f);
  }

  return n;
}
```

Que valores de teste para n precisaremos para cobrir todas as declarações? Dois valores para n funcionarão da seguinte forma: n menor que 0

e n maior que 0. Os dois valores fornecem cobertura de ramificação? Não, pois precisamos testar n igual a zero para cobrir a ramificação “nunca iterar o loop”. Os três valores garantem cobertura de condições? Sim, nesse caso, pois não há condições compostas no programa. Que tal cobertura de loop? Bem, não exatamente. Teremos que cobrir n igual a 1 e n igual a qualquer que seja o número máximo de iterações.

Que valor para n usamos para obter cobertura de loop? O que aconteceu com o fatorial no valor máximo?

Resultados obtidos nos testes:

- **n = 12**
- **12! = 479001600**
- **n = 13**
- **13! = 1932053504**
- **No entanto, a resposta real para 13! é 6.227.020.800**

Isso é um bug? Se entendemos como os computadores armazenam números inteiros, certamente veremos como o valor inteiro de f teria problemas de “overflow”.

Esse exemplo ilustra um perigo do teste de caixa branca, não especificamos os resultados esperados com antecedência. A saída pode parecer razoável, mas está correta? Precisamos cuidar com os testes de unidade de caixa branca sem critérios confiáveis para nos ajudar na determinação do status de aprovação/reprovação do teste.

Para que realmente possamos implementar o teste de caixa branca, é importante que entradas, resultados e especificações estejam bem descritos e elaborados, considerando ainda o critério de aceite para a certificação da qualidade do código.

TEMA 2 – TIPOS DE TESTES: TESTE RELACIONADO COM A MUDANÇA

Testes relacionados com a mudança são realizados para a verificação de mudanças em um sistema ou um aplicativo, o que pode afetar outras partes do sistema de maneira indesejada. São executados para garantir a integridade do sistema e minimizar o risco de problemas em áreas não relacionadas à versão original.

Geralmente, esses testes são realizados quando o software sofre algum tipo de alteração grande, o que pode ocasionar um número de bugs considerável. Precisamos reexecutar um roteiro de teste, criado a partir de um teste funcional (de regressão). Melhoramos a eficiência desse tipo de teste ao automatizá-lo. Ainda conseguimos garantir a redução de custos na execução, o que torna a verificação mais rápida.

Esse tipo de teste pode ser realizado após as mudanças implementadas. O objetivo concentra-se nos possíveis efeitos indesejados que as alterações podem causar em qualquer parte não modificada. Pode ainda incluir testes de unidade, integração, regressão e outros tipos que visam garantir que o sistema continue funcionando de maneira eficiente e correta após a mudança.

Os testes relacionados à mudança são importantes porque ajudam a identificar problemas rapidamente e corrigi-los antes que causem problemas maiores ou impactem negativamente os usuários finais. Também ajudam a garantir a qualidade e a confiabilidade do sistema, o que é fundamental para manter a satisfação dos clientes e garantir o sucesso do projeto.

Testes de regressão funcionais podem utilizar ferramentas, como no caso de Selenium WebDriver. Eles também podem ser aplicados com técnicas estruturais (caixa-branca), pois a prática de execução de testes que validam o software em funcionamento é chamada de regressão.

Dentro da mesma lógica, segundo a qual um teste relacionado à mudança deve considerar o código novo e o código legado, vamos aplicar todos os demais testes já conhecidos, entre os quais destacamos:

- **Testes unitários:** pequenos e isolados, cobrindo funcionalidades limitadas, enquanto os testes de integração juntam os testes unitários dentro de uma funcionalidade interdependente. Os principais pontos fortes dos testes unitários referem-se ao fato de serem pequenos e rápidos para que sejam executados. Como cada teste é limitado, precisamos muitos deles, o que é aceitável, porque eles são rápidos de escrever e executar.
- **Testes de integração:** ajudam a desacoplar diferentes partes dos projetos. Se uma equipe concluiu o trabalho em seu módulo, podemos testar esse trabalho isoladamente para ganhar confiança, mesmo que outro trabalho necessário, em outro lugar, ainda não esteja concluído. Precisamos de mais testes do sistema quando ambas as seções

estiverem concluídas, ainda que os testes possam começar antes disso. Os testes de integração correspondem às unidades funcionais do código, facilitando a identificação de quem é o responsável pelos problemas encontrados. Se uma única equipe tiver esse módulo, ela precisará corrigir os problemas que forem encontrados. As falhas são mais rápidas de diagnosticar, porque vêm de um único módulo. Assim, são mais fáceis de depurar do que os testes de sistema, em que vários módulos trabalham juntos.

- **Testes de regressão:** são quase tão extensos quanto os testes abrangentes, mas incluem apenas os testes que desejamos executar quando esse recurso for modificado. Embora não existam alterações de código, não precisamos executar todos os testes. Depois dos testes de regressão, temos os testes manuais/noturnos, com um limite de tempo (por exemplo, 12 horas para uma execução noturna). Portanto, eles precisam ser priorizados com cuidado. Encontram problemas que podem afetar usuários ativos, em pequenas interrupções, e abrangem casos que consomem muito tempo para que sejam incluídos nos testes de CI/CD.

O que importa em um teste relacionado às mudanças é que ele continue garantindo a qualidade do software. Inevitavelmente, vamos trabalhar com testes estruturais (caixa branca), testes funcionais (caixa preta), testes não funcionais e finalmente, em sua completude, testes relacionados com mudança.

TEMA 3 – NÍVEIS E TIPOS DE TESTES

Um sistema de software geralmente é composto de vários subsistemas, que por sua vez são compostos de vários componentes, geralmente chamados de “unidades” ou “módulos”. A estrutura do sistema resultante também é chamada de “arquitetura de software” ou “arquitetura do sistema”. Projetar uma arquitetura que suporte perfeitamente os requisitos do sistema é uma parte crítica do processo de desenvolvimento de software.

Durante o teste, um sistema deve ser examinado e testado em cada nível de sua arquitetura, desde o componente mais elementar até o sistema completo e integrado. As atividades de teste relacionadas a um determinado nível da arquitetura são conhecidas como um “nível” de teste. Cada nível de teste é uma única instância do processo de teste.

A seguir, vamos detalhar as diferenças entre os vários níveis de teste com relação aos seus diferentes objetos de teste, considerando objetivos de teste, as técnicas de teste e as responsabilidades/funções (Spillner, 2007):

- **Teste de componente:** o teste de componentes envolve a verificação sistemática dos componentes de nível mais baixo na arquitetura de um sistema. Dependendo da linguagem de programação usada para criá-los, esses componentes apresentam vários nomes, como “unidades”, “módulos” ou (no caso de programação orientada a objetos) “classes”. Os testes correspondentes são, portanto, chamados de “testes de módulo”, “testes de unidade”, “testes unitários” ou “testes de classe”.
- **Componentes e testes de componentes**: independentemente da linguagem de programação usada, os blocos de construção de software resultantes são os “componentes”, enquanto os testes correspondentes são chamados de “testes de componentes”.
- **Base de teste**: os requisitos específicos do componente e o projeto do componente (ou seja, as suas especificações) devem ser consultados para formar a base de teste. Para projetar testes de caixa branca ou avaliar a cobertura de código, devemos analisar o código-fonte do componente e usá-lo como uma base de teste adicional. No entanto, para julgar se um componente reage corretamente a um caso de teste, devemos consultar a documentação do projeto ou dos requisitos.
- **Objetos de teste**: como vimos, módulos, unidades ou classes são objetos de teste típicos. Aspectos como shell scripts, scripts de banco de dados, procedimentos de conversão e migração de dados, conteúdo do banco de dados e arquivos de configuração também podem ser objetos de teste.
- **Um teste de componente verifica a funcionalidade interna de um componente**: um teste de componente normalmente testa apenas um único componente, isolado do resto do sistema. O isolamento serve para excluir influências externas durante o teste. Se um teste revelar uma falha, é obviamente atribuível ao componente que estamos testando. Também simplifica o design e a automação dos casos de teste, por conta de seu escopo restrito. Um componente pode apresentar vários blocos de construção. O importante é que o teste do componente verifique apenas a funcionalidade interna do componente em questão, não a sua interação com componentes externos a ele. Esse último é objeto de teste de

integração. Os objetos de teste de componentes geralmente chegam “frescos do disco rígido do programador”, de modo que esse nível de teste é muito próximo do trabalho de desenvolvimento. Os testadores de componentes, portanto, precisam apresentar habilidades de programação adequadas para que o trabalho seja feito corretamente.

- **Testes de desenvolvedor**: para escrever um driver de teste, precisamos de habilidades de programação. É muito importante estudar e entender o código do objeto de teste (ou pelo menos sua interface) para programar um driver de teste capaz de chamar corretamente o objeto de teste. Em outras palavras, precisamos dominar a linguagem de programação envolvida, garantindo acesso às ferramentas de programação apropriadas. É por isso que o teste de componentes geralmente é realizado pelos próprios desenvolvedores do componente. O teste é muitas vezes chamado de “teste de desenvolvedor”, mesmo que “teste de componente” seja o significado real.
- **Teste versus depuração**: testes de componentes geralmente são confundidos com a depuração. No entanto, a depuração envolve eliminação de defeitos, enquanto o teste envolve a verificação sistemática de falhas no sistema. A depuração pode ser considerada um processo em que sabemos que algo “quebrou”, e assim precisamos consertar o “defeito”. Geralmente, iniciamos o processo detectando o defeito. Depois, simulamos a execução passo a passo, de forma manual, até encontrar o problema. Normalmente, o trabalho é feito através de ferramentas ou extensões para a IDE que estamos trabalhando. Quando comparamos testes com depuração, estamos falando de testes que são realizados por analistas de testes de forma manual ou automatizada, pré-definidos e desenhados para encontrar falhas na programação. A depuração, em geral, é feita pelos desenvolvedores; porém, com a evolução de técnicas, métodos e ferramentas para testes de software, hoje encontramos no mercado de trabalho analistas de testes que operam muito bem a depuração de código.
- **Funcionalidade de teste**: a tarefa mais importante de um teste de componente é verificar se o objeto de teste implementa, completa e corretamente, a funcionalidade definida em suas especificações. Esses testes também são conhecidos como “testes de função” ou “testes

funcionais". Nesse caso, a funcionalidade equivale ao comportamento de entrada/saída do objeto de teste. Para verificar a integridade e a correção da implementação, o componente é submetido a uma série de casos de teste, cada um cobrindo uma combinação específica de dados de entrada e saída.

- **Teste de eficiência**: o atributo de um teste de eficiência indica como, economicamente, um componente interage com os recursos de computação disponíveis. Isso inclui aspectos como uso de memória, uso do processador ou tempo necessário para executar funções ou algoritmos. Ao contrário da maioria dos outros objetivos de teste, a eficiência de um objeto de teste pode ser avaliada com precisão a partir de critérios de teste adequados, como kilobytes de memória ou tempos de resposta medidos em milissegundos. O teste de eficiência raramente é realizado para todos os componentes de um sistema. Geralmente, fica restrito a componentes que apresentam determinados requisitos de eficiência definidos no catálogo de requisitos ou na especificação do componente. Por exemplo, recursos de hardware limitados disponíveis em um sistema embarcado, ou um sistema de tempo real que precisa garantir limites de tempo de resposta predefinidos.
- **Teste de manutenibilidade**: a manutenibilidade incorpora todos os atributos que influenciam na facilidade ou dificuldade de aprimorar ou estender um programa. O fator crítico aqui é a quantidade de esforço necessária para que um desenvolvedor obtenha uma compreensão do programa existente, considerando ainda o contexto. Isso é válido tanto para um desenvolvedor que precisa modificar um sistema que programou anos atrás, quanto para alguém que está assumindo o código de um colega. Os principais aspectos de manutenibilidade que precisam ser testados são: estrutura de código, modularidade, comentários de código, compreensibilidade e atualização da documentação.
- **Estratégias de teste**: o teste de componentes é altamente orientado para o desenvolvimento. O testador geralmente tem acesso ao código-fonte, suportando uma técnica de teste orientada para caixa branca em testes de componentes. Nesse caso, um testador pode projetar casos de teste usando o conhecimento existente da estrutura interna, além de métodos e variáveis de um componente.

- **Testes de caixa branca**: a disponibilidade do código-fonte também é uma vantagem durante a execução do teste, pois você pode usar ferramentas de depuração apropriadas para observar o comportamento das variáveis durante o teste e verificar se o componente funciona corretamente. Um depurador também permite manipular o estado interno de um componente, para que você possa iniciar exceções deliberadamente quando for testar a robustez. Durante a integração, os componentes individuais são combinados, cada vez mais, em unidades maiores. As unidades integradas podem ser muito grandes para inspecionar o código completamente. Se o teste de componente é feito em componentes individuais ou em unidades maiores (compostas de vários componentes), temos aqui uma decisão importante que deverá ser tomada como parte do processo de integração e planejamento de teste.
- **Testar primeiro**: abordagem de ponta para testes de componentes (cada vez mais, também em níveis de teste mais altos). A ideia é primeiramente projetar e automatizar os casos de teste, para programar o código que implementa o componente em uma segunda etapa. Essa abordagem é fortemente iterativa: você testa o código com os casos de teste que já projetou e, em seguida, estende e melhora o código do produto em pequenas etapas, repetindo o processo até que o código cumpra os seus testes. Esse processo é chamado de “programação test-first” ou “desenvolvimento orientado a testes”, muitas vezes abreviado para TDD (“Test-Driven Development”). Se derivamos os casos de teste sistematicamente, usando técnicas de design de teste bem fundamentadas, essa abordagem produz ainda mais benefícios. Por exemplo, testes negativos também serão elaborados antes que você comece a programar, e assim a equipe será forçada a esclarecer o objetivo pretendido, considerando o comportamento do produto para esses casos.
- **Teste de integração**: o teste de integração é o próximo nível do teste de componente. Assume que os objetos de teste entregues a esse nível já são componentes testados e que todos os defeitos internos dos componentes foram corrigidos, na medida do possível.

Desenvolvedores, testadores e equipes de integração especializadas montam grupos desses componentes em unidades maiores. Esse processo é

chamado de “integração”. Depois de montados, devemos testar se os componentes integrados interagem corretamente entre si. Esse processo é conhecido como “teste de integração”, com o objetivo de encontrar falhas nas interfaces e na interação entre os componentes integrados.

Nesse nível, todos os documentos que descrevem a arquitetura de software e o projeto do sistema de software, especialmente especificações de interface, fluxo de trabalho e diagramas de sequência, além de diagramas de casos de uso, devem ser consultados como base de teste.

Podemos nos questionar por que o teste de integração é necessário quando todos os componentes já foram testados individualmente. Nosso estudo de caso ilustra os tipos de problemas que precisam ser resolvidos.

O teste de componentes não pode garantir que as interfaces entre os componentes estejam livres de falhas. Isso é o que torna o nível de teste de integração crítico para o processo geral de teste. Falhas potenciais de interface devem ser descobertas, e suas causas isoladas. Quando as interfaces para sistemas externos de software (ou hardware) são testadas, o processo geralmente é chamado de “teste de integração em geral” ou “teste de integração de sistema”.

O teste de integração do sistema só pode ser executado após a conclusão do teste de sistema. Nessa situação, corremos o risco de que a equipe de desenvolvimento só será capaz testar a “sua metade” da interface em questão. A “outra metade” é desenvolvida externamente; portanto, pode mudar inesperadamente a qualquer momento. Mesmo que os testes do sistema sejam aprovados, isso não garante que as interfaces externas sempre funcionem conforme o esperado.

Isso significa que existem vários níveis de teste de integração, que por sua vez abrangem objetos de teste de tamanhos variados (interfaces entre componentes internos ou subsistemas, ou entre todo o sistema e sistemas externos, como serviços da web, ou entre hardware e software). Se, por exemplo, os processos de negócios devem ser testados na forma de um fluxo de trabalho de interface cruzada, composto por vários sistemas, pode ser extremamente difícil isolar a interface ou o componente que causa a falha.

O processo de integração monta componentes individuais para produzir unidades maiores. Idealmente, cada etapa de integração é seguida por um teste de integração. Cada módulo assim construído pode servir como base para uma

maior integração, em unidades ainda maiores. Os objetos de teste de integração, portanto, podem consistir em unidades (ou subsistemas) que foram integradas iterativamente.

Na prática, os sistemas de software raramente são construídos do zero, pois são resultado da extensão, modificação ou combinação de sistemas existentes. Muitos componentes do sistema são produtos padronizados comprados no mercado aberto. O teste de componentes geralmente não inclui esses tipos de objetos existentes ou padronizados, enquanto o teste de integração deve cobrir os componentes e a sua interação com outras partes do sistema. Os objetos de teste de integração mais importantes são as interfaces que conectam dois ou mais componentes. Além disso, o teste de integração pode abranger programas e arquivos de configuração. Dependendo da arquitetura do sistema, o acesso a bancos de dados ou outros componentes de infraestrutura também pode fazer parte do processo de teste de integração do sistema.

O teste de integração também exige que os drivers de teste forneçam dados aos objetos de teste, coletando e registrando os resultados. Como os objetos de teste são unidades compostas que não apresentam interfaces com o mundo externo, que ainda não sejam fornecidas por suas partes componentes, faz sentido reutilizar os drivers de teste criados para os testes de componentes individuais.

Se a etapa de testes de componentes foi bem organizada, você terá acesso a um test driver genérico para todos os componentes, ou pelo menos um conjunto de test drivers que foram desenhados segundo uma arquitetura unificada. Se for esse o seu caso, os testadores podem adaptar e usar os drivers de teste existentes para testes de integração com um mínimo de esforço extra.

Um processo de teste de componentes mal organizado talvez forneça apenas alguns drivers adequados com estruturas operacionais diferentes. A desvantagem desse tipo de situação é que a equipe de teste agora precisa investir muito tempo e esforço na criação ou na melhoria do ambiente de teste, em um estágio avançado do projeto, desperdiçando um tempo precioso de teste de integração.

Como as chamadas de interface e o tráfego de dados por meio das interfaces do driver de teste precisam ser testados, o teste de integração geralmente usa “monitores” como ferramenta de diagnóstico adicional. Um

monitor é um programa que verifica a movimentação de dados entre os componentes e registra o que vê. O software comercial está disponível para monitorar o tráfego de dados padrão, como protocolos de rede, enquanto você terá de desenvolver monitores personalizados para uso com interfaces de componentes específicos do projeto.

O objetivo do teste de integração é encontrar falhas na interface. Certos problemas podem ocorrer durante a primeira tentativa de integração de dois componentes se os formatos de interface não correspondem, se os arquivos necessários estiverem faltando, ou ainda se o desenvolvedor tiver componentes programados que não seguem as divisões especificadas. Tais falhas geralmente serão detectadas logo no início, como falhas na compilação ou nas execuções de compilação.

Em qual sequência os componentes devem ser integrados para maximizar a eficiência do teste? A eficiência do teste é medida por meio da análise da relação entre os custos do teste (equipe, ferramentas e assim por diante) e a utilidade (número e gravidade de falhas descobertas) para um determinado nível de teste. O gerente de teste é responsável por escolher e implementar a estratégia ideal de teste e a integração para o projeto em questão.

Componentes individuais são finalizados em semanas ou meses. Gerentes de projeto e gerentes de teste não desejam esperar até que todos os componentes relevantes estejam prontos para que sejam integrados em uma única execução. Uma estratégia simples para lidar com essa situação é integrar os componentes na sequência (aleatória) em que são finalizados. Isso envolve verificar se um componente recém-chegado deve ser integrado a um componente ou subsistema já existente. Se essa verificação for bem-sucedida, o novo componente pode ser integrado e a integração testada.

TEMA 4 – TESTES NÃO FUNCIONAIS

Os testes não funcionais verificam várias características de um software. Não precisamos compreender as regras do negócio. Tais testes complementam os testes funcionais e também outros.

Os requisitos não funcionais descrevem os atributos do comportamento funcional de um sistema, ou seja, a qualidade com que um sistema ou componente deve cumprir a sua função. A sua implementação influencia

amplamente a satisfação do cliente/usuário e, portanto, também a forma como o sistema é apreciado (Spillner, 2007).

Do ponto de vista do fabricante, flexibilidade e portabilidade são aspectos importantes da manutenibilidade de um produto, pois ajudam a reduzir os custos de manutenção. As seguintes características não funcionais do sistema devem ser cobertas e verificadas pelos testes correspondentes (geralmente, durante o teste do sistema):

- **Carga:** medição do comportamento de sistema sob carga crescente (por exemplo, número de usuários paralelos e número de transações).
- **Desempenho:** medição da velocidade de processamento e tempos de resposta para casos de usos específicos, geralmente em conjunto com o aumento da carga.
- **Volume de dados:** observação do comportamento do sistema, dependente de volumes de dados (por exemplo, ao processar arquivos muito grandes).
- **Estresse**: observação do comportamento do sistema em situações de sobrecarga.
- **Segurança de dados:** combate ao sistema não autorizado e/ou acesso a dados.
- **Estabilidade/Confiabilidade:** em uso constante (por exemplo, medição do número de falhas do sistema por hora para perfis específicos de usuário).
- **Robustez:** quando submetido a erros do usuário, com erros de programação, falha de hardware e similares. Testando o tratamento de exceções e o comportamento de reinicialização/recuperação.
- **Compatibilidade/conversão de dados**: testar a compatibilidade com sistemas existentes, especialmente durante a importação/exportação de dados.
- **Diferentes configurações**: por exemplo, usando diferentes versões de sistema operacional, idiomas ou plataformas de hardware.
- **Usabilidade**: testar a facilidade de aprendizado e a simplicidade de uso, incluindo a compreensibilidade da saída para vários grupos de usuários.

- **Conformidade da documentação do sistema com o comportamento do sistema:** por exemplo, manual do usuário versus GUI ou descrição da configuração versus comportamento real.
- **Manutenibilidade:** compreensibilidade e atualização da documentação de desenvolvimento, com estrutura modular e assim por diante.

Os requisitos não funcionais são frequentemente formulados de forma incompleta ou vaga. Atributos como “o sistema precisa ser fácil de usar” ou “resposta rápida” não podem ser testados em sua forma atual. Os testadores devem participar das revisões da documentação dos requisitos, certificando-se de que todos os requisitos (não) funcionais listados sejam redigidos de forma a torná-los mensuráveis – portanto, testáveis.

Além disso, muitos requisitos não funcionais são ostensivamente tão óbvios que ninguém pensa em mencioná-los ou especificá-los. Tais “requisitos assumidos” são frequentemente altamente relevantes, e assim o sistema deve apresentar os atributos implícitos correspondentes.

TEMA 5 – TESTE DE MANUTENÇÃO

A manutenibilidade incorpora todos os atributos que influenciam a facilidade (ou dificuldade) de aprimorar ou estender um programa. O fator crítico aqui é a quantidade de esforço necessária para que um desenvolvedor (equipe) obtenha uma compreensão do programa existente e de seu contexto. Isso é válido tanto para um desenvolvedor que precisa modificar um sistema que programou anos atrás quanto para alguém que precisa assumir o código de um colega (Aniche, 2022).

Os principais aspectos de manutenibilidade que precisam ser testados são: estrutura de código, modularidade, comentários de código, compreensibilidade e atualização da documentação.

A forma como o SUT (System Under Test - Sistema em Teste) pode ser testado por meio de automação de teste depende em grande parte da forma como pode ser gerenciado e monitorado por meio de interfaces. A testabilidade e a automatização do SUT são fatores importantes no sucesso de um projeto de automação de teste. Embora a testabilidade seja igualmente importante para cenários de teste automatizados e manuais, a automatização se concentra na compatibilidade do SUT com a automação de teste em geral.

A testabilidade de um SUT precisa ser avaliada e verificada. “Testabilidade” é uma subcaracterística da característica de qualidade de software padrão ISO “Manutenção”. Trata-se de uma característica não funcional que pode ser abordada em um estágio inicial do processo de design do SUT. É responsabilidade dos arquitetos e desenvolvedores garantir que o SUT possa ser testado da forma mais fácil possível. No entanto, essas funções geralmente carecem de um ponto de vista de teste específico. Idealmente, os TAEs (Test Automation Engineers – engenheiros de automação de testes) são envolvidos no início do projeto do SUT, para que venham a fornecer informações valiosas sobre a testabilidade geral do sistema. Em geral, a interação entre a ferramenta de automação de teste e o SUT ocorre por meio de interfaces. A adequação dessas interfaces, com relação aos seguintes fatores, determina em última instância a testabilidade do SUT:

- **Controle do SUT:** os testes automatizados usam interfaces para acionar ações e eventos dentro do SUT. Isso é feito, por exemplo, por meio de APIs, protocolos de comunicação, elementos de interface do usuário ou comutadores eletrônicos.
- **Observabilidade:** testes automatizados usam interfaces para verificar se o comportamento real do SUT corresponde ao comportamento esperado.
- **Arquitetura clara:** para obter uma estratégia de automação de teste consistente e transparente, você precisa definir claramente quais interfaces estão disponíveis para a automação de teste, em qual nível de teste. O grau de intrusão e os efeitos da ferramenta de automação no SUT também devem ser avaliados.

Quanto mais fácil for implementar os três fatores listados, melhor será a testabilidade do SUT. Se novas interfaces forem necessárias, ou se as existentes precisarem ser estendidas, é tarefa do TAE especificá-las e comunicá-las aos desenvolvedores em um estágio inicial, para que o esforço necessário possa ser considerado em tempo hábil.

É sempre importante lembrar que interfaces de teste novas e modificadas também precisam ser testadas – em outras palavras, análises estáticas, revisões e testes dinâmicos também são necessários.

A variedade de interfaces de testes que levam a potenciais melhorias de testabilidade é enorme. Vejamos alguns exemplos:

- Interfaces de teste para monitorar e controlar o SUT, por exemplo, se nenhuma interface gráfica do usuário estiver ainda disponível.
- Testar interfaces para determinar o estado atual do SUT quando um teste baseado em estado é realizado.
- Espaços reservados ou stubs que simulam software e/ou hardware.
- Interfaces/placeholders/stubs/drivers que simulam modos de falha (por exemplo, falha de hardware), usando ferramentas de injeção de falhas.
- Script de cálculos e planilhas.
- Ao avaliar a testabilidade e sua otimização potencial, por meio do fornecimento de ganchos de teste adicionais, garantimos que os testes automatizados possam ser executados de forma eficaz e eficiente. É interessante notar que o teste manual, que pode ocorrer em paralelo, também se beneficia da testabilidade aprimorada.

Os seguintes fatores também são importantes quando tratamos de automatização:

- A compatibilidade com as ferramentas de teste existentes deve ser garantida em um estágio inicial.
- A compatibilidade da ferramenta de teste é crítica, pois pode afetar a capacidade de automatizar o teste de funções importantes.
- Soluções para melhores recursos de automação podem exigir o desenvolvimento de código personalizado e chamadas para APIs.

A manutenção de software é inevitável, até porque um software é algo dinâmico, que sofre mudanças constantes, sejam elas por melhorias ou por correções.

FINALIZANDO

A definição e a escolha de testes dependem de cada projeto em que estamos trabalhando, considerando ainda a filosofia de trabalho em termos de qualidade de nossa empresa e o orçamento que temos disponível para a aplicação dos testes necessários e adequados.

Mais do que classificar e criar dependências teóricas entre tipos diferentes de testes, precisamos compreender o que é necessário testar, verificando e validando os processos em cada momento do desenvolvimento do software, quando ele vai para produção e mesmo depois, quando está em produção, a

depender de mudanças e outras situações inesperadas.

A escolha por testes no momento da integração (unitários, contratuais ou integração), no momento da entrega (integração novamente, end-to-end ou performance), ou no momento em que vai para produção (Smoke, Health, Monitoring e Alerts), é o que mais nos importa.

Precisamos conhecer os vários níveis e tipos de testes para aplicá-los, de forma inteligente, em cada etapa do processo de desenvolvimento de software, considerando ainda a sua manutenção.

REFERÊNCIAS

ANICHE, M. **Effective Software Testing**. Shelter Island: Manning Publications, 2022.

BLACK, B. R. **Pragmatic Software Testing**: Becoming An Effective And Efficient Test Professional. New Jersey: Wiley, 2007.

SPILLNER, A. **Software Testing Practice**: Test Management. Sebastopol: O'Reilly Media, Inc., 2007.